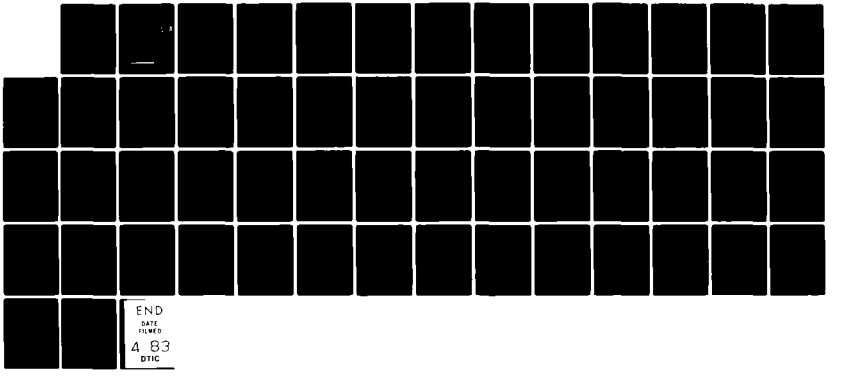
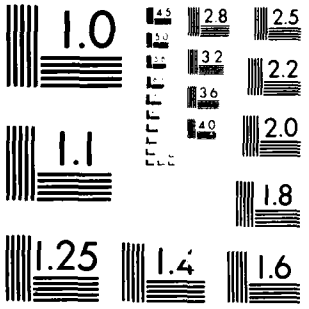
AD-A126 560 REQUIREMENTS FOR THE PROTOCOL LABORATORY AND TEST FACILITY(U) SYSTEM DEVELOPMENT CORP SANTA MONICA CA 04 DEC 81 SDC-TM-7038/214/00 DCA100-80-C-0044 1/1

UNCLASSIFIED F/G 5/1 NL



END
DATE FILMED
4 83
DTIC



This document was produced by
System Development Corporation in performance of Contract DCA100-80-C-0044

TM a working paper

System Development Corporation
2500 Colorado Avenue • Santa Monica, California 90406

series base no./vol./release
TM- 7038 /214/00
author SYTEK Staff
technical Carl M. Switzky
release David J. Kaufman
for Charles A. Savant
date 12/4/81

DCEC PROTOCOLS STANDARDIZATION PROGRAM

REQUIREMENTS FOR THE PROTOCOL LABORATORY AND TEST FACILITY

ABSTRACT

As part of a general program promoting controlled development and introduction of new DoD protocols, a Protocol Laboratory is to be built by the Defense Communications Engineering Center. This document specifies requirements for the laboratory, including an interactive development environment for protocol specifications, validation tools, and an implementation testbed.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 7038/214/00	2. GOVT ACCESSION NO. AD A126560	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Requirements for the Protocol Laboratory and Test Facility		5. TYPE OF REPORT & PERIOD COVERED interim technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Sytek Staff		8. CONTRACT OR GRANT NUMBER(s) DCA100-80-C-0044
9. PERFORMING ORGANIZATION NAME AND ADDRESS System Development Corporation 2500 Colorado Ave. Santa Monica, CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 33126K Task 1053.558
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Center Switched Networks Engineering Directorate 1860 Wiehle Ave., Reston, VA 22090		12. REPORT DATE 4 Dec 81 13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) N/A		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18. SUPPLEMENTARY NOTES

This document represents results of interim studies which are continuing at the DCEC of DCA.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Protocols, Data Communications, Data Networks, Protocol Standardization, Protocol Laboratory & Testing.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

As part of a general program promoting controlled development and introduction of new DoD protocols, a Protocol Laboratory is to be built by the Defense Communications Engineering Center. This document specifies requirements for the laboratory, including an interactive development environment for protocol specifications, validation tools and an implementation testbed.

DD FORM 1 JAN 73 1473 EDITION OF 1 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

As part of a general program promoting controlled development and introduction of new DoD protocols, a Protocol Laboratory is to be built by the Defense Communications Engineering Center. The laboratory will be used in essentially three ways:

1. as a general tool supporting basic research in protocol engineering issues,

2. as a protocol development facility for the specification, validation, and testing of new protocols prior to their designation as a DoD standard, and

3. as a central certification site for acceptance testing of non-laboratory protocol implementations, supporting DCA's "global" (i.e. multi-site) configuration control of network software.

The second and third uses of the laboratory are of primary concern, and are closely coupled, requiring the development of common tools to manage a protocol throughout its entire "life-cycle". This effort focuses on standardization via carefully written protocol specifications and implementations developed within the controlled laboratory environment.

Future distributed systems can be thought of as consisting of three types of components: hardware, software, and protocols. Like the other components, protocols have their own life-cycle of conception, design, specification, implementation, and maintenance. It is important to recognize that the protocol life-cycle is long; protocols will likely outlive all hardware and software components in the distributed system. For this reason, the development of correct and efficient protocols is crucial, and premature standardization on the wrong protocols must be avoided. The Protocol Laboratory can play a critical role in the DoD standardization effort, helping to ensure that the proposed standards meet the necessary requirements, are valid, and can be efficiently implemented.

This document specifies requirements for the fully developed laboratory, including an interactive development environment for protocol specifications, validation tools, and an implementation testbed. No attempt has been made here to describe a realistic implementation schedule. However, interdependence of laboratory components, alternative approaches and their relative technical feasibility are discussed whenever appropriate.

Section 2 of this document presents the general model of protocol development and life-cycle management which the laboratory is to support. This includes a discussion of "traditional" protocol development, "top-down" development, and a development model which recognizes the benefits of iterative protocol design and implementation using proper management tools.

Section 3 gives an overview of the protocol laboratory, describing its components and general procedures for their use.

Sections 4 to 8 present the detailed requirements for the individual tools.

Finally, Section 9 summarizes the laboratory requirements.

2. THE PROTOCOL DEVELOPMENT PROCESS

Before a protocol development facility can be designed, a general development philosophy must be adopted. Protocol development will typically contain elements of the following general activities:

- o initial conception and general placement within a protocol architecture
- o service requirements definition, leading to a service specification
- o study of the feasibility of various technology or design options
- o design of the protocol mechanisms, leading to a mechanism specification
- o validation of mechanism conformance to the service specification
- o initial implementations
- o testing of initial implementations
- o "release" of specifications for production implementations
- o certification of implementations

These development activities are discussed in the next section, followed by an analysis of three protocol development methodologies which differ in their approach to activity scheduling.

2.1 PROTOCOL DEVELOPMENT ACTIVITIES

2.1.1 Initial Conception and Architectural Placement

The initial concept for a new protocol may arise from a variety of sources, including new user requirements and new technological opportunities. Such a new protocol will not exist in isolation - it must be coherently integrated into an environment of existing and proposed protocols.

Thus the first step in a new protocol's development is an analysis of its proper placement within the protocol architecture. Such an analysis will clarify the services the new protocol should provide, and will also identify the services it can build upon which are provide by other network elements.

Throughout this report, we assume the existence of a well defined architecture - independent of any particular protocols - which provides a context for new protocol development [1]. Specification efforts within the protocol laboratory can then proceed with a well-understood "immediate environment" in which the new protocol is to be embedded.

2.1.2 Service Specification Development

A service specification is a description of what services the protocol is to provide to the next higher layer in the protocol architecture. This

specification does not include a description of how the protocol goes about providing the given services. Various methods of presenting service specifications can be used, such as natural language text or a more formal specification language.

Service specifications can be checked for syntax errors, self-consistency, and faithful representation of the intended design. Such validation of a service specification depends on the specification language in use. The DoD protocol specification guidelines [2] call for both an informal, natural language description of services and a formal service/interface specification. This formal specification involves an abstract machine interface which defines the manner in which the protocol responds to user service requests.

2.1.3 Mechanism Specification Development

A mechanism specification is a description of the "internal" behavior of the protocol interpreter, including the rules it follows in communicating with peer entities. Formal techniques for mechanism specifications can be based on non-procedural specification languages, programming languages or state machines. The DoD specification guidelines call for an approach based on the concept of an extended state machine.

As with the service specification, the mechanism specification must be checked for self-consistency. However, in addition to testing the mechanism specification's self-consistency, we must be assured that the mechanism specification conforms to the service specification.

2.1.4 Development of Initial Implementations

Once a set of protocol mechanisms have been designed, initial implementations can be built and tested, both for conformance to a service specification, and to determine the protocol's performance characteristics.

A useful analogy can be made between the testing of a "conventional" product and a new protocol. In the former case, in-house or "alpha" testing as it is commonly known is performed on the product to shake out initial bugs. Alpha test products are used by "internal" customers who can provide quick feedback on problems and provide real usage experience before the product is released. After this, the product is sent to "beta" test sites where customers are given first access to a new product in return for being test sites. Of course, in the case of protocol development it is the protocol itself (i.e. the specifications) which must be alpha-tested, and the software implementation is just a tool used during the tests.

Under this analogy, the protocol development process should include at least one and possibly multiple implementations for testing. It is important that such implementation testing be performed under a wide variety of network environments.

It must be emphasized that such testing is not intended to "shake down" a particular implementation; rather, the goals are to:

1. determine performance-relevant aspects of the protocol's design,

2. expose mechanisms which are difficult to implement or are ambiguously specified, and

3. validate conformance to the service specification.

The initial implementations could be used to provide the protocol designer with valuable feedback, and may not necessarily be suitable as "production" implementations of the protocol.

2.1.5 Development and Certification of Production Implementations

The ultimate goal of the protocol development process is the installation of implementations "out in the field". These implementations will generally be developed by software designers working from the protocol specifications, without access to the code or development tools produced for the initial implementations. In fact, implementations in a variety of languages running on a variety of machines can be anticipated.

The formal acceptance of each new field implementation will require a battery of tests. The lack of a controlled environment may preclude the testing of the implementation's mechanisms in the same manner in which the "alpha" implementations were tested. However, it is possible to "service test" an implementation using a test driver at the implementation's site, in conjunction with another "already certified" implementation.

Current service testing of such implementations generally consists of "bake-offs" between two such programs. The most manageable approach to such bake-offs would use a "standard" implementation as the test partner. For this to work, the test coverage must be so comprehensive that interoperability with the standard will guarantee interoperability with any other certified implementation. Such a guarantee is difficult to achieve in practice, but nonetheless the use of a standard could greatly simplify the total certification process. Since the standard implementation exists within a controlled environment, comprehensive tests can be applied uniformly to all new implementations.

The standard implementation of a protocol must be developed within a controlled environment to ensure that it fully conforms to the specifications. One approach is for the standard implementation to be at least partially built via automated techniques directly from the mechanism specification.

An acceptable performance against the standard may not in fact guarantee acceptable performance against other field implementations. Such anomalies would indicate a problem with the standard or with the test procedures, and corrective measures could then be taken. Ideally, one would need only to validate an implementation's conformance to the standard to be assured of its complete system-wide compatibility.

2.2 PROTOCOL DEVELOPMENT METHODOLOGIES

This section examines three basic protocol development methodologies: the traditional "ad hoc" approach, the "top down" method, and finally an iterative protocol design and implementation model. All three methodologies have as their ultimate goal the production of fully specified protocol design; they differ in how that goal is reached. We will argue below for the iterative approach as being the most suitable within the Protocol Laboratory.

2.2.1 The Traditional Approach

The traditional approach to protocol development does not initially include any formal statement of need or formal service specification activities. Rather, the protocol is concurrently designed and implemented with only very general requirement guidelines and the service provided is specified by after-the-fact descriptions. This "design by implementation" approach has significant merit since the design and implementation are intimately related, and therefore aspects of the design that are difficult or inefficient to implement can be immediately changed. Also the problem of writing realistically implementable specifications is solved. This method has its disadvantages, however.

First, it is really only viable for small projects that can be designed and built by a very small number of people. Second and more importantly, since the service specification is not formalized, the protocol has no formal design goals, and its behaviour is a byproduct of the implementation rather than a requirement of the design. This can result in complex mechanisms and inconvenient interfaces tailored to a specific implementation environment. Finally, without a service description the concept of being "done" or of the protocol being "complete" is undefined and the design task is potentially unbounded.

Perhaps the most significant problem with such an ad hoc approach to protocol development stems from the multi-environment nature of a computer network. Typically a protocol must be implemented in a variety of hardware/software environments, in contrast to other software systems which exist in a single implementation (though perhaps multiple instances). Such single-environment software systems are often built paying only lip-service to their original specifications, a situation not acceptable for protocol development. A protocol specification must be completely independent of any particular implementation, and will continue to play a major role even after several implementations are completed. This is the major failing of "design by implementation" when applied to protocols.

2.2.2 The Top Down Approach

The top-down approach to protocol development attempts to solve some of the problems of the previous approach by making the design task more structured. First a formal service specification is defined which describes the protocol's functions, and its user and low level interfaces. Next, a mechanism specification is designed which implements the service specification. Finally one or more implementations are built.

This solves the problems which arise due to the lack of a service specification, and helps ensure a desirable user interface. However, by rigorously separating the specification, design, and implementation tasks, the implementation consequences of specific design decisions are not obvious.

Implementation considerations such as complexity, efficiency, and buffer utilization are very important, yet these are not formally factored into the mechanism design phase. Rather, the implementation characteristics are "discovered" after the complete mechanism is designed, and performance is measured only after implementation is completed. In some instances, the mechanism specification may be released prematurely before a test implementation has been created, and the design's implementation characteristics studied. Once released, it is very difficult to change a design since many implementations may be underway or may be complete before a design change decision can be made.

This approach is also not conducive to design changes (iterations) that are guaranteed to occur. Since the full mechanism is designed as a unit, without implementation reviews, protocol sub-mechanisms can easily become heavily interlocked making it difficult to modify one sub-mechanism without affecting the integrity of the whole system. An alternative approach which aids design modularity is outlined in the next sub-section.

2.2.3 The Iterative Model for Protocol Development

The iterative model recognizes the utility of feedback between the design, specification, and implementation phases of a protocol's development, provided that such feedback can be structured to occur in a controlled environment.

As previously mentioned, a service specification is essential to know what a protocol should do and to check an implementation for conformance. Formal service specifications are typically not written before a protocol is designed because they are difficult to write. This difficulty arises from the lack of a priori knowledge of costs and tradeoffs of the protocol's various features. Recognizing this, it is proposed that the service specification be developed iteratively, with successive versions building on experience gained from the previous ones. For each service specification version, a new mechanism specification and implementation version could be constructed. The task of deciding when the design task is complete has therefore been transformed into deciding whether the latest service specification is acceptable. This method, coupled with a good configuration control system, allows the development process to backtrack if necessary and can provide comparisons between versions with different mechanisms providing the same feature.

The idea is to provide a tighter coupling between design decisions made in specifications and their implementation consequences, and to ensure that at any phase in the development cycle, there is a consistent set of specifications for any implementation.

Within this methodology, it is possible to perform iterative protocol design as follows. The development could start with a simple service definition that contains "core" features of the desired service and basic performance

characteristics. For example, for a transport protocol, the core features may be those required for reliable data exchange. Given this, an initial mechanism specification is designed and tested for conformance to the service specification. Finally, an implementation is designed and tested. Once this is done, additions and modifications to the protocol will begin at the service specification and travel through to a modified implementation. This way, the implementation effects of additions to either specification can be immediately seen.

The goal of such controlled iteration is the effective management of a protocol's development through its shake-down period prior to "release". By using the laboratory to develop both specifications and an initial implementation, a protocol's designers can be sure that further iterations (based on implementation considerations) will not be necessary after the protocol's specifications are received by non-laboratory implementers. Whatever iterations are necessary should occur within the laboratory under strict engineering management. (Of course, this is an ideal, and further iterations may always occur during a protocol's life-cycle).

The modularity within a protocol's design which is necessary if such an iterative approach is to succeed may not be natural for some protocols. However, it seems that such modularity is one distinguishing characteristic of higher level protocols (beyond transport). Since these protocols will in fact be the primary focus of the protocol laboratory, it is reasonable to concentrate on the iterative approach to protocol development. In addition, the DoD protocol specification guidelines call for the use of "hierarchical state machines", which will allow specification and implementation of well defined submachines. Thus the iterative approach to protocol development is consistent with the specification techniques to be used.

The protocol laboratory's tools must be designed to support this iterative refinement, allowing specifications and implementations to be modular; useful features include explicit support for "don't care" clauses and default modules. Strong configuration management within the laboratory - on specifications as well as implementations - is crucial if such iterations are to be controlled.

2.3 SUMMARY - PROTOCOL DEVELOPMENT VIA A LABORATORY

Requirements for the protocol laboratory arise from the approach to protocol development discussed above.

The laboratory must support the production and analysis of protocol service and mechanism specifications. It must support the creation of initial protocol implementations within the laboratory, and must provide the means for their testing under a variety of network characteristics. It must provide a controlled environment for the development of standard protocol implementations, which can be used to certify other implementations developed external to the laboratory. It must provide a certification facility and support the administration of certification procedures.

Iterations between the specification and initial implementation phases of a protocol's development must be allowed and even encouraged within the laboratory environment. The goal is the production of stable protocol specifications which, upon release from the laboratory, allow separate external teams to easily build working and efficient implementations.

3. OVERVIEW OF THE PROTOCOL LABORATORY

The laboratory will be required to simultaneously support many distinct tasks. Here we only describe the mainline laboratory activity involving new protocol development and external implementation certification.

3.1 LABORATORY ACTIVITIES

Although the laboratory's tools must fit coherently into a unified system, it is possible to separate them into a set of "subsystems". Each subsystem consists of a set of related tools and procedures providing support for a particular activity within the laboratory, namely:

1. Protocol Development - the laboratory includes an interactive environment supporting the development of protocol designs and specifications

2. Protocol Analysis - a set of computation-intensive specification analysis and validation tools

3. Implementation Development - compilers and other tools providing a software development environment for the creation of protocol implementations

4. Implementation Testing - a testbed of target machines in a flexible network-emulation environment allowing realistic functional and performance testing of protocol software

5. External Implementation Certification - tools and administrative procedures supporting the certification of protocol implementations developed outside the laboratory

6. Laboratory Management - a laboratory-wide system of configuration control and other administrative functions

The following sections present a brief overview of each of these activities and the systems required for their support. The remainder of this report will present detailed requirements for the laboratory components, tools, and procedures.

3.1.1 Protocol Development

A protocol is typically modeled as an interpreter which changes state and takes action in response to external stimuli. DoD protocols will be specified using a specification technique which formalizes this state-machine approach.

The Protocol Laboratory will include an interactive environment which allows the protocol designer to specify a protocol state-machine, stimulate it with simulated external events, monitor its behavior, and then edit its specification as desired. This protocol development system is based on the machine-interpretable DoD specification language, and allows the designer to "walk-through" the specification. Since the system knows the required syntax for the state-machine specification, syntax-directed editing will be possible.

This will encourage good specification habits and will ease the designer's text manipulation burden. Such a facility is similar in some respects to many current program development systems (e.g. for LISP) which include interpreters, syntax-directed editors, and other tools in a uniform environment.

Protocol service specifications will also include machine-interpretable text. The protocol development system will include support for the creation of service simulators based on such a service specification. This will allow full simulation of peer-protocol interactions, using two (or more) protocol state-machine interpreters together with a service simulation of their lower layer.

The intent of the interactive protocol development system is neither to produce final specifications nor to produce implementations; it will, however, support both activities by ensuring that the fundamental mechanisms defined in the protocol are well-understood.

3.1.2 Protocol Analysis

The protocol development system described above automates (to some extent) the current process of "desk-checking" a new protocol specification. However, such automated walk-throughs cannot guarantee that subtle design errors are not present in the protocol. Deadlocks or unreachable states are examples of problems which may not be detectable without a thorough formal analysis of the specification. Such analyses typically involve intensive computations.

Specification analysis tools which "guarantee" the validity of a protocol are not technically feasible at present. However, the laboratory will provide the tools necessary to investigate possible approaches to protocol validation and eventually to develop a full analysis system.

3.1.3 Program Development

It is known that protocol performance can be extremely sensitive to implementation details and the underlying network characteristics. DoD standard protocols must not include mechanisms which lead to unacceptable performance in critical situations. It is also necessary to ensure that a protocol is clearly specified in an unambiguous fashion, allowing separate implementation teams to work independently and still produce interoperable products. For these reasons, the laboratory must include facilities for the development and testing of actual protocol implementations.

The program development environment at the laboratory must include compilers and other tools. The program developers will generate implementations based on specifications produced with the protocol development tools.

3.1.4 Implementation Testing

The protocol implementations developed within the laboratory are not intended to be "production" implementations suitable for distribution to external sites (though that may in fact happen). Rather, their purpose is to provide protocol developers with early implementation feedback which may impact the ultimate design.

Implementations should be tested in as "real" an environment as possible. This implies that the laboratory must provide a "testbed" in which multiple implementations on separate processors communicate via a network. The network must be capable of exhibiting a variety of delay and error characteristics, allowing the implementations to be tested under conditions anticipated to exist in the field.

Testing must be isolated from other laboratory activities. Consequently the testbed processors should be distinct from the implementation development system. However, control and monitoring of the tests will involve other processors besides the testbed target machines.

3.1.5 External Implementation Certification

Eventually a protocol specification will be released to the external (non-laboratory) world, perhaps with a "DoD Standard" stamp. Implementations based on this specification will be built for a wide variety of underlying hardware/software environments. Such implementations must be certified.

The laboratory can support the certification process by providing an implementation against which other implementations can be tested. This implementation can be derived from the laboratory-internal implementation, and test procedures used within the laboratory during protocol development can be adapted for the certification task. External access to the laboratory is desirable for such testing; other tests and administrative procedures can augment this process to provide a complete certification program.

3.1.6 Laboratory Management

The laboratory must support the concurrent design, specification, implementation, and testing of multiple protocols. This requires a set of management tools for configuration control, access control, scheduling and resource allocation.

Configuration control within the laboratory goes beyond simple code configuration control. During the course of its development within the laboratory, a protocol will evolve through multiple service specifications, mechanism specifications, implementations, test plans, test results, and other supporting documents. It must always be ensured that the correct relationships are maintained among the various versions of the documents. For example, it must be clear which test result demonstrates conformance between which version of the specification and which version of the code.

In addition, configuration control must be maintained over the laboratory tools themselves, which will evolve over the lifetime of the lab: this is a general administrative problem independent of any one protocol's development. Also, the laboratory should provide configuration control support for the "global" distribution of protocol implementations after their certification.

Other administrative tools are necessary to control access to the laboratory from remote sites. In general, the laboratory will be a "closed system" - however, it must be accessed remotely for three distinct purposes:

1. Informal access to protocol documentation, mail service, etc.

2. Remote access to laboratory tools by contractor or government personnel during protocol development (however, the bulk of this work of course must occur in the lab)

3. access to the tester subsystem for certification of an external implementation

The laboratory should provide access control mechanisms allowing the administration to properly schedule and protect laboratory resources. Such mechanisms will also be required for internal laboratory operations involving the simultaneous development of multiple protocols.

3.2 SKETCH OF LABORATORY CONFIGURATION

Based on the above discussion of laboratory activities and the systems required for their support, an "ultimate" laboratory configuration can be laid out. A broadband cable local communication network could be used to connect all the elements of the laboratory into a well integrated system; elements of the laboratory would communicate with one another over separate, frequency isolated channels. This configuration is shown in Figure 1.

The requirements for the initial laboratory configuration could be a subset of that envisioned for the ultimate configuration. Until such time as required, the functions of the two general purpose computer systems could be provided by a single unit (the "Development System"). External access (for mail, documents, etc.) could also be limited. Terminals attached to the development system, via connection to the laboratory local cable network, could provide the user workstation facilities. Target machines (2) could attach to the local cable network, communicating with the development system over channels separate from each other and from those used by the workstations. External implementation access and specialized hardware for graphics and hardcopy would not be needed initially.

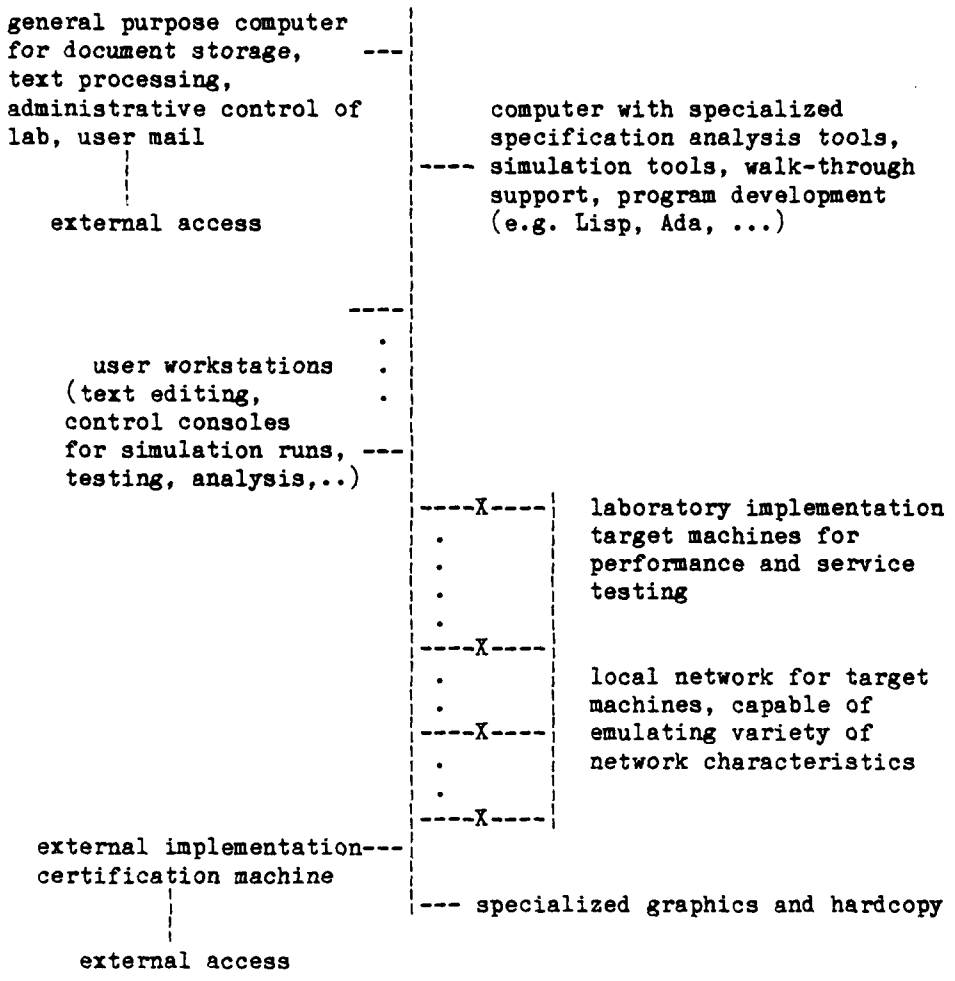


Figure 1: Laboratory Configuration

4. PROTOCOL DESIGN AND SPECIFICATION

One of the primary roles of the Protocol Laboratory is as a provider of a friendly development environment for the protocol designer. Here "development" refers to development of protocol specifications, not development of implementations. Functional consistency and completeness of a protocol's mechanisms are the initial goals of the designer; the protocol laboratory should support in a natural way the activities necessary to achieve these goals.

4.1 OVERVIEW: AN INTERACTIVE PROTOCOL DEVELOPMENT ENVIRONMENT

The primary focus of the protocol developer is the design of a state machine for a protocol interpreter. A comprehensive interactive development environment for the protocol designer can be provided within the laboratory by emphasizing the "interpreter" aspect of his product.

A protocol interpreter "interprets" externally generated events (user commands, timeouts, arriving data, ...) by acting in a manner dependent upon its current state. The protocol designer must lay out the set of possible states, events, and the actions to be taken in response to the events. Current protocol design efforts involve frequent manual walkthroughs, during which the protocol designer "deskchecks" his paper design. To check the interactions of peer interpreters, two identical state machines must be simultaneously walked-through. Automated support of this process can greatly enhance the protocol designer's work environment.

Given a state-machine description language such as that proposed for the DoD protocol specification technique, a general interpreter can be built. A user interface to this interpreter can allow the specification of a set of states, events, and actions defining a particular protocol. Once the protocol is defined and entered into the system, the designer can step through its states by presenting sequences of events to the interpreter. Such an interactive environment would free the protocol designer from the bookkeeping aspects of a protocol walk-through.

A syntax-directed editor can be incorporated into this interactive environment to allow the user to easily edit the data structures which together define the protocol. Such editors are common in LISP environments [3] and have also been built for a number of block-structured languages [4]. Pretty-printing and syntax enforcement are typical of such an editor.

Separate from the protocol's mechanism specification will be a service specification. This specification will also be in a formalized language, which will allow the construction of a service simulator for a protocol. A service simulator for a lower layer can then be used in conjunction with the state-machine interpreter to provide automatic support for a full walk-through of the protocol's peer interactions.

With a complete implementation of the interactive protocol development system, a user session could proceed as follows:

1. Sitting down at his terminal, the user invokes the protocol development system, which prompts him for a command.

2. The user restores a previously saved environment, which may include a protocol specification, an event file, or perhaps tailored commands.

3. By invoking the editor, the user makes some changes to a specification. The editor enforces some syntax constraints.

4. Leaving the editor, the user initializes a single protocol interpreter and "plays" events against it. At various points the user asks for a display of state information, undoes a previous event, or repeats an event. Here the user is artificially generating all events, including timeout and lower layer interface events.

5. Detecting an error in his specification, the user reinvokes the editor and corrects it - saving the state of his walkthrough for immediate resumption upon leaving the editor.

6. Satisfied with behavior of his protocol under the "single interpreter" walkthrough, the user establishes a "peer interpreter" walkthrough by invoking a lower layer service simulator. During the course of this walkthrough, the interactive protocol development system is maintaining state information on two interpreters communicating through a simulated lower layer. The user can display state information for both protocol state machines, and can monitor (or initiate) events.

7. Exiting the walkthrough, the user saves the current protocol specification and logs off for the day.

By using canned files of events sequences, the use of such an interpreter could best be described as a "symbolic execution" of the protocol.

Of course, the activities supported by this interactive system are just one phase of the complete protocol development process; in particular, one does not gain real implementation experience (or performance data) from the tools described in this section.

For both the service and mechanism specifications, it is likely that the protocol designers will use a combination of "ordinary" text processing tools and the interactive tools discussed above. For example, initial concepts may be first written using a standard text-editor for circulation among development team members; these ideas will then be formalized and refined using the interactive tools; finally, when a "production specification" is to be created, the formal protocol descriptions will have to be incorporated into a document which includes accompanying text.

The next two sections describe the facilities of the protocol development system in more detail, focusing on the interactive tools.

4.2 SERVICE SPECIFICATIONS

4.2.1 Language

Service specifications are to be used for three distinct purposes which place different requirements on the specification language:

1. as a guide to the protocol user, describing what services are provided; for these purposes natural language text (augmented where possible with more "formal" descriptions) provides the proper balance between easy understandability and non-ambiguity,

2. as a statement of requirements which are to be placed on the mechanism specification, from which appropriate tests can be derived; a formal language which allows nondeterministic behavior to be expressed is required, and

3. as a foundation for the construction of a "service simulator" which can be used to test higher-level protocols; this requires a language which explicitly identifies relevant parametrizations of nondeterministic service behavior.

It seems unlikely that a single service specification language can be developed in the near term which will meet all the above requirements. Separate "images" of a protocol's service specification will of necessity be developed for a particular purpose. The DoD protocol specification guidelines [2] explicitly require both an informal description of services in natural language and a formal service/interface specification. The protocol laboratory must provide tools which when properly used will ensure consistency among these images.

4.2.2 Service Specification Development Process

Development of a protocol's service specification within the laboratory proceeds through five phases:

Phase 1: Creation of an initial version of "Services Provided to the Upper Layer" (Section 2 of the total protocol specification as required by the Guidelines). This is an informal, natural language description and would be created using ordinary text-processing tools. This initial document will serve to generate comments and will evolve as the protocol development proceeds.

Phase 2: Development of a formal (machine readable) interface specification which defines a particular format for the interface events and specifies precisely the manner in which nondeterministic aspects of the interface can be parametrized. This work is performed within the interactive protocol development environment, which includes a syntax-directed editor for the formal service specification syntax. The specification created here will ultimately evolve into Section 3 of the final protocol specification ("Upper Layer Service/Interface Specifications").

Phase 3: Working from both the informal service description produced in Phase 1 and the Phase 2 formal service specification, a set of service tests can be generated. Initially such test cases will be manually produced from the specifications - later laboratory support may include automated service test generation tools.

Phase 4: Integration with a service simulation system which takes as input the formal interface specification produced in Phase 2, a script which assigns a probability distribution to the nondeterministic aspects of the service, and a sequence of test events from Phase 3. The output of the simulation runs contains the interleaved sequences of "service request primitives" and "service response primitives".

Phase 5: Analysis of the simulation runs, which may result in a return to an earlier phase.

The manually generated and automatically generated tests are used here only to validate that the service behavior as codified in the service specification is as intended by the protocol designers. Later these same tests will be used to validate the mechanism specification and actual implementations of the protocol.

Of course, in keeping with the fundamental "iterative" approach to protocol development taken by the laboratory, the service specifications created by the above five-phase process may in fact undergo revision based on the results of the mechanism development effort or the experience gained in the implementation testing.

The final "production" service specification will be created using ordinary text-processing tools, suitably incorporating the formal service/interface specification.

4.2.3 Tools

Laboratory tools supporting the various phases must be coordinated into a general configuration control scheme for the service specification development.

4.2.3.1 Tools for Phase 1

The Phase 1 support consists of ordinary text-processing tools allowing the development of the "Services Provided to the Upper Layer" section of the protocol specification.

The use of these tools for this purpose does not differ from their use for general text processing. However, the document produced must be administered by the laboratory-wide configuration control system as discussed later in this document.

4.2.3.2 Tools for Phase 2

The primary tool for the development of a formal service/interface specification is a syntax-directed editor for the specification language. This requires that the grammer of the service specification language be expressed in a machine-readable form; an editor can then be developed which "knows" the language.

Such an editor could support the specification writer in the following ways:

1. With appropriate prompts, the editor can guide the user through the necessary format of the specification, e.g. declarations come first.

2. Simple syntax errors can be caught in an interactive mode by the editor, rather than requiring a compilation process which may fail due to trivial misspellings.

3. Searches within an editing session can be tailored to the language syntax rather than being based on pure string comparisons.

4. The editor can "prettyprint" the specification, with a user-selectable "nesting depth" option. Prettyprinting displays the text in a meaningful way; a user-selectable depth option allows the high-level structure of the current specification to be displayed, with ellipsis used to hide irrelevant low-level details.

Such an editor will also be important in the creation of protocol mechanism specifications; in fact, a common baseline editor can be tailored for both purposes by parameterizing it with the two separate specification grammers.

4.2.3.3 Tools for Phase 3

The generation of service test cases will be a manual process in the initial laboratory. Support for this procedure may be limited to an ordinary text editor with which files of test event sequences can be created. Additional automated test case generation (working from a test case "template") could also be provided.

These test files will be administered by the laboratory-wide configuration control system.

4.2.3.4 Tools for Phase 4

The laboratory contains a general-purpose protocol service simulator, which takes as input:

- a formal interface specification as produced in Phase 2

- a script identifying probability distributions to be assigned to the non-determistic aspects of the interface specification

- a test event sequence consisting of user-generated "service request primitives"

Configuration control for multiple simulation runs (with corresponding simulation results) ys provided by the laboratory.

The service simulation facility will be used for two purposes: first, to check that the service specification is as desired, and secondly during the development of a higher protocol's mechanism specification as part of a walkthrough of the higher protocol.

4.2.3.5 Tools for Phase 5

Analysis of the simulation runs will require "post-processing" tools which abstract relevant behavior from the raw output. Graphics support for such analysis may be desired.

4.3 MECHANISM SPECIFICATION

The laboratory will support the writing and analysis of a protocol's mechanism specification via the tools described in this section.

4.3.1 Language

The mechanism specification will be expressed in a language which formalizes the "augmented state machine". As with the service specification, the mechanism specification is used for several distinct purposes which place different requirements on its actual format. First, to be a truly useful guide to implementers, the mechanism specification must ensure clarity through the liberal use of natural language constructs. However, for the second purpose of analysis and validation it is desirable to replace such informal language with a fully formal specification.

The mechanism specification will exist in at least these two versions: the "approved for release" version which will be distributed to implementers, and the "laboratory internal" version which is used for formal validation. The differences between the two must be minor and essentially syntactical. It must always be clear that the abstract machines specified are identical.

4.3.2 Mechanism Specification Development Process

Development of a protocol's mechanism specification within the laboratory proceeds through six phases:

Phase 1 Creation of the "Overview of Protocol Mechanisms" for Section 6.1 of the total protocol specification as required by the guidelines. This is in natural language text.

Phase 2 Development of a machine readable version of the mechanism specification, defining a particular format for interface events. This development starts with the specification produced in Phase 1 as a baseline, and will evolve to form Sections 6.1 (Message Formats) and 6.2 (Extended State Machine Representation) of the protocol specification.

Phase 3 Working from both the natural language specification of Phase 1 and the formal specification of Phase 2, sequences of test events are listed which will be used for "mechanism testing" of the protocol.

Phase 4 Automated walk-throughs of the state machine providing preliminary self-consistency and reachability validation of the mechanism specification. Most of the effort here will be an informal, interactive process on the part of the individual protocol designer; later, more formal walk-throughs can be conducted using the mechanism test sequences generated in Phase 3.

Phase 5 Service testing via symbolic execution of multiple state machines with service simulation of lower layer protocols.

Phase 6 Analysis of test results in preparation for iteration.

4.3.3 Tools

The laboratory tools supporting the creation of a mechanism specification form the heart of the interactive protocol development environment. A general-purpose state-machine interpreter, which can take a protocol state machine specification as input and then be "executed" interactively as part of a walk-through, is the most important piece of this support system.

4.3.3.1 Tools for Phase 1

Phase 1 of the protocol's mechanism specification involves only ordinary text-processing tools for natural language. The files produced here must be administered by the laboratory's configuration control system.

4.3.3.2 Tools for Phase 2

Phase 2 requires use of a syntax-directed editor, with requirements similar to those discussed earlier for a service specification editor. In fact, both editors can be built on a common base, tailored for a particular language by including its grammer specification in a parametrized manner.

The syntax-directed editor for mechanism specifications must be capable of cooperating appropriately with the interactive state-machine interpreter. This involves editing state machine descriptions in the middle of an interpretive walk-through.

4.3.3.3 Tools for Phase 3

Protocol mechanism testing involves the exercising of a state machine with control over event occurrences at all three interfaces (upper, lower, and system). Generation of appropriate test event sequences can be either manual or automatic. Initially the Protocol Laboratory could support manual mechanism test sequence generation using ordinary editing tools. Automated test generation will be discussed in a later section of this document covering Protocol Analysis, as the tools required are similar in nature to the tools necessary for automated protocol validation.

4.3.3.4 Tools for Phase 4

A general purpose interactive state machine interpreter will be a major tool of the Protocol Laboratory.

The interactive environment will include a command interpreter. Users can define a state machine (states, events, actions) either by invoking the syntax-directed editor or by restoring a previously saved machine definition. The state machine definition will be in the formal language used for a protocol's mechanism specification as required for Sections 6.2 and 6.3 in the specification guidelines.

Once the state machine has been defined to the current environment and initialized, the user can "walk-through" its behavior in response to user-generated events. For example, these events could be commands from the higher level protocol, system events such as timeout occurrences, or received messages from the lower protocol interface. Predefined event sequences could be used during a session to quickly move through those state transitions which are not of current concern.

User activities during such a walk-through would be similar in many respects to those within a debugging session using an interactive symbolic debugger. At various points the user would examine current state information. Actions taken by the state machine in response to a prior event would be displayed. "Breakpoints" could be set to return control to the user command interface when a particular state is entered, or a particular event arises. The hierarchical nature of the state machine specification technique could be used to advantage to incorporate previously "debugged" submachines into the interactive walk-through.

The user should be given the ability to invoke the specification editor at any time during the session. A reinitialized machine could then be executed; alternatively, the remainder of the walkthrough could proceed from its prior state using the newly-edited sections of the machine's specification. Such use of the editor must be done in a careful fashion, but could be used to great advantage by a knowledgable user.

It must be possible to record the results of such a walkthrough for later analysis. This could be as simple as a "photo" facility which places the user's terminal input and output on a file such as in Tops-20; this is only possible for scroll-mode terminals. Since the interactive walkthrough could use terminal cursor addressing to great advantage, a more complex recording strategy may be necessary.

Informal walkthroughs of the protocol will form a part of the protocol developer's day-to-day work; event sequences used in such walkthroughs can be conceived on the fly. More formal walkthroughs can proceed using the test event sequences generated in Phase 3 above.

4.3.3.5 Tools for Phase 5

The walkthroughs performed during Phase 4 involve the behavior of a single protocol interpreter. Such a walkthrough implicitly includes the behavior of a peer interpreter in the form of events at the lower level interface. Phase 5 walkthroughs will explicitly incorporate a peer interpreter; two peer state machines will "communicate" through a service simulation of the lower level protocol.

Service simulators were discussed earlier in this report. Phase 5 walkthroughs could proceed informally (with the user generating both upper layer events and system events); more formal "service testing" could proceed using an emulation of the protocol's surrounding system (its "execution environment" as required for Section 7 of the protocol's specification) in addition to the lower layer service simulation. Such an effort would use test sequences

generated during the development of the protocol's service specification, consisting solely of events occuring at the upper layer interfaces of the two peer interpreters. This would in effect be a complete "symbolic execution" of the protocol in a peer-to-peer configuration.

4.3.3.6 Tools for Phase 6

Analysis of the test results (both for the "single interpreter" and the "peer interpreters" walkthroughs) may require specialized tools tailored to the individual protocol. Requirements for such analysis tools cannot be at present defined; however, the test output files and the test analysis results must be maintained under configuration control in conjunction with the specifications.

5. PROTOCOL SPECIFICATION ANALYSIS

The previous section discussed requirements for an interactive protocol development environment. Such an interactive environment would provide automated support for the otherwise manual process of "walking through" a protocol's specification. By encouraging the frequent checking of a protocol's mechanisms during its design phase, the interactive environment will yield a better "product".

However, if we are to have complete confidence in a protocol's design, a more thorough analysis is required than can be achieved through walk-throughs. Tools supporting such analyses fall into two categories:

1. tools for the analysis of a single protocol interpreter's behavior ("State Machine Analysis")

2. tools for the analysis of the interactions of peer interpreters ("Peer Interaction Analysis")

The following two sections describe requirements for analysis tools which could be implemented for the Protocol Laboratory.

5.1 STATE MACHINE ANALYSIS

A protocol specification should meet certain very general requirements which have nothing to do with the specific services of the individual protocol. For example, every state must be reachable from the initial state via some sequence of events. Similarly, it is generally not a good idea for a state machine to "hang" in one state for the occurrence of all possible external events, unless the state is a desired termination state. These are requirements on a "well formed state machine", and the fact that protocol state machines normally act in a peer relationship is irrelevant.

This section discusses tools for state machine analysis that could be incorporated into the Protocol Laboratory.

5.1.1 The Problem with Extended State Machines

In a "pure" state machine specification, the necessary analyses are easy so long as the number of states is small. A state's reachability can be verified simply as follows: "mark" the initial state; mark those states which are reachable from states already marked, repeating until no new states are marked; check if the desired state is marked.

This procedure is not computationally feasible if the number of states and/or events is large. Unfortunately, this is typically the case with protocols, where one is faced with a problem commonly referred to as "state explosion" - the number of states required is too large to allow a pure state machine specification. The state explosion problem is dealt with in the "extended" state machine specification technique through the use of variables and predicates. The use of variables and predicates makes it easier to write and understand the specification; unfortunately it does not simplify the reachability analysis problems inherent in a state-rich machine.

Similarly, it is easy in a pure state machine (with few states and events) to check for a hanging state: one merely looks at each state and make sure that some event causes a state transition. In an extended state machine specification, the states are determined by the values of a set of variables, and different events are given by the values of various event parameters. The number of checks involved will be large; this would of course be eased if one could initially throw out unreachable states.

Such analysis of an extended state machine cannot be based on "exhaustive search" techniques appropriate for pure state machines. If the analysis technique is to avoid exhaustive searches, it must use the way in which the states and events are organized (into variables, parameters, and predicates). This requires considerable intelligence on the part of the analysis algorithms, as will become clear in the following sections.

5.1.2 Reachability/State Exploration via Backtracking

Simple reachability analysis does not make sense for a protocol extended state machine. First, due to the number of states and the manner in which they are organized, it may be perfectly acceptable for some states to be unreachable; some combinations of variable values may not play any role in the protocol's definition. Secondly, it is not enough just to know that a state can be reached - typically it is necessary to determine that the state can be reached following a specific sequence of transitions in response to a specific sequence of events.

The DoD protocol specification technique calls for the use of a "main scalar element which is commonly referred to as the 'state' and a set of additional 'state variables'" [2]. Through the use of such a primary state variable, the most fundamental aspects of the protocol's behavior can be easily described. It is certainly desirable to analyze the reachability of these primary states; the more thorough reachability analysis of every combination of state variable values cannot be practically achieved.

Typically the protocol designer has in mind a set of important sequences of transitions between primary states. These are the mainline transition sequences which occur in response to the normal event sequences. For example, the sequence "closed - listen - openReceived - established" may be considered fundamental in a transport protocol. Given such an "intended" transition sequence, the protocol designer will want to answer the following question: "Does the protocol as specified really allow these transitions to occur in the intended way?" This is often not a simple question, as the actual transitions involved may require that complex relationships hold between the primary state variable, other non-primary state variable values and the values of event parameters.

The protocol designer wants to determine which sequences of events (and which assignments to the event parameters) will drive the protocol extended state machine through a given sequence of transitions. This is the converse of the information obtained via a walkthrough, which shows the state transitions taken in response to a given sequence of events. The interactive environment which provides automated support for walking through a protocol specification can be viewed as a set of tools which allow the protocol designer to execute the machine in its ordinary "forwards" direction. In contrast, to determine which event sequences will drive the machine through a given transition sequence, it is necessary to drive the machine "backwards".

For our purposes, a mechanism specification can be viewed as a set of transitions between primary states. Each transition is labeled with:

1. the event which triggers it (including event parameters),

2. an enabling predicate (involving event parameters and various state variables) which must be true for the transition to occur, and

3. a set of actions which take place along with the primary state transition.

The actions may include, for example, assignments to state variables. The DoD specification technique uses Ada fragments to specify actions.

Suppose that it has been determined that a particular sequence of transitions is to be analyzed. For example, the intent of the protocol designer may be that a particular primary state is normally reached only by following a specific sequence of transitions. To validate that the "goal" state is indeed reachable via that transition sequence, it is necessary to determine which event sequences (if any) will cause the desired transitions.

Each transition involves an event with its event parameters, and an enabling predicate giving a condition on the event parameters and variables which must be satisfied for the transition to occur. We face the following problem: what values should be given to the event parameters to ensure that the transitions actually will occur? How can we determine those parameter values which actually make the desired enabling predicates true?

Stated more formally, our problem is as follows: let a transition be described with the notation

T = <S, D, EV(P1 .. Pn), EP, A>

where S is the source state, D is the destination state, EV(P1 .. Pn) is the triggering event with its parameters, EP is the enabling predicate, and A is the associated action. Then given a sequence of transitions T1, T2, ... Tm we must determine values for the various parameters of EV1, EV2, ... EVm which will make the predicates EP1, EP2, ... EPm true.

The problem is complicated by the fact that an action associated with one transition may affect the value of a later transition's enabling predicate. Since the actions taken may depend on the values of the event parameters, this implies that the necessary parameter values cannot be determined on a "transition-by-transition" basis. In fact, a particular value for one parameter (chosen to make a particular enabling predicate true) may preclude the satisfaction of a later enabling predicate. The sequence of enabling predicates must be "solved" for parameter values, taking into consideration the effect of the various actions. This is analogous to backtracking through the given transition sequence to determine the necessary events and event parameters. The determination of these events cannot in general be accomplished through a "forward" execution of the state machine (since the necessary event parameters are not known in advance).

The "solving" of event parameters for a given transition sequence involves the symbolic manipulation of the associated enabling predicates. In general, these symbolic manipulations are computationally difficult but are well within the state of the art. In practice, it seems that the manipulations required for real protocols will be relatively simple for the most part.

The result of such symbolic manipulations of enabling predicates for a given transition sequence will be a set of simultaneous conditions on the event parameters involved. These will consist of, for example, equations and inequalities governing the relationships which must hold between the event parameters in order for the designated sequence of transitions to occur. There are two possible outcomes:

1. The set of conditions have a solution (i.e. they are consistent). This implies that the "goal" primary state is indeed reachable via the intended sequence of transitions.

2. The set of conditions do not have a solution (i.e. they are inconsistent). This implies that the designated transition sequence is impossible, and an unreachability situation has been exposed.

The second possible outcome requires some elaboration. In this case, there is no set of assignments to the relevant event parameters which will allow the designated sequence of transitions to occur. It could be that at first glance it seems plausible that the sequence is possible - it is only after a detailed analysis of the event parameters and the enabling predicates that its impossibility is established. If this transition sequence is intended to play a role

in the protocol's operation, a flaw in the protocol's specification has thus been identified.

An automatic tool performing the backtracking necessary for such reachability analyses would operate as follows:

- Input: the protocol's mechanism specification and a designated sequence of transitions between primary states.

- Processing: starting in the last state of the transition sequence, conditions on the events and event parameters are determined which are necessary for the given transition to occur. This requires an analysis of the enabling predicates and actions associated with the given transition. This is repeated backwards through the sequence of transitions, successively generating additional conditions on the events. Finally, the conditions are "solved" to generate a final set of conditions on the sequence of events.

- Output: a sequence of events (with associated values for the event parameters) which if "played" against the extended state machine would force it through the designated transition sequence.

Solving the conditions could be semi-automatic, requiring human help if an impasse is reached.

5.1.3 Test Generation for Implementations

The same tools described above for reachability analyses can be used to generate test sequences which check an implementation's conformance to the mechanism specification. Although this activity falls under the protocol's implementation phase, we discuss it here due to the possible sharing of tools with the analysis phase.

During the mechanism specification phase of a protocol's development, the design team validated the correctness of the proposed mechanisms using the laboratory's walk-through support provided in the interactive protocol development environment. These validation efforts use automated tools which are essentially driven directly off the machine-readable version of the mechanism specification.

The first real implementation will likely use some of the routines developed during the mechanism validation process, but the direct relationship between the specification and the state machine's execution will no longer exist. Since the experience gained during the testing of this first implementation may affect the specification of the protocol, it is imperative that its conformance to the specification be validated. This requires a development environment which allows close interaction with the specification analysis tools.

In particular, the development environment should allow mechanism testing of the implementation in a "test chamber", with all interfaces under control of the test software. Similar mechanism testing was performed in the symbolic

execution effort - but there the emphasis was on checking the consistency and completeness of the protocol's mechanism specification. In contrast, mechanism testing of the implementation is oriented toward validating its conformance to a specification which is assumed correct.

The implementation must be embedded in an environment which allows test software to control all three interfaces (upper, lower, system) as called out in the mechanism specification. Such an environment can consist of predefined, universally-applicable software modules which generate and monitor test event sequences, which are bound to the protocol code at compile-time. Such a process would be similar to compilation with a debug option invoked, with the debugger here being a highly specialized system tailored to the testing of protocol software which has been based on a highly structured specification. The test software could then control the execution of the protocol code.

After passing such mechanism testing, the implementation can be ported to the network-based test environment for service testing and performance experimentation under a variety of realistic network conditions.

The tools which generate test event sequences for conformance testing of the first implementation are the same as those used during the reachability analyses and state exploration. Using a machine readable mechanism specification in the form of an extended state machine, event sequences (with appropriate parameter values) are generated which will drive the protocol through a designated sequence of state transitions. The output of a reachability analysis effort can thus be reused as test sequences for an implementation.

The development machine will include a family of compilers, differing only in the final code generation for the target machine. These compilers will allow a rich choice of compile-time options which generate code tailored to a particular testing mission. One of these compilers will generate code native to the development machine, and will include a compile-time option which will bind the protocol code with specialized conformance-testing software. This software will "control" the execution of the protocol code by managing the event occurrences across the upper, lower, and system interfaces. This will allow "playing" the test event sequences generated previously and checking the behavior of the implementation via its interface actions. This is similar to compiling with a debug option, where the debugger is particular to protocol software. One can envision the setting of "breakpoints" which halt execution of the protocol whenever a non-conforming event occurs, reporting such to a terminal and awaiting further instruction from a human operator.

5.2 PEER INTERACTION ANALYSIS

The above analysis of a protocol specification did not depend on the fact that the analyzed object exists naturally in a set of peers. The reachability analysis tools would work on any extended state machine specification, whether or not it defined a protocol. Here we discuss tools which are particular to protocols, analyzing the peer interactions.

5.2.1 Deadlock Detection

Protocol deadlock situations fall into two classes.

A complete deadlock can occur if a pair of protocol entities both enter states which can only be exited upon an event occurrence which depends upon the other's actions. If such actions cannot occur, no progress will be made. (We discuss only deadlocks involving a pair of peers - multipeer deadlocks are conceptually similar).

Partial deadlocks are similar, but the states may be exited upon elapse of a timer. Such timer events are often included in a protocol specification precisely to avoid complete deadlock situations. However, the action taken is often to abort the current activity. A partial deadlock differs from a complete deadlock in that some progress will be made, but it is not the desired progress.

Of course, escape from complete deadlocks via timer events is necessary in certain situations, for example to protect against network vagaries. However, the protocol designer will want to avoid partial deadlocks which may cause abortions even when the network behavior is acceptable.

The laboratory may provide tools which support the analysis of a protocol specification for complete or partial deadlocks. Again, the state explosion problem makes an exhaustive search technique computationally infeasible. However, analysis for deadlocks involving the extended state machine's primary states may be possible.

The basic tool would differ little from the "automated walkthrough" support described earlier to check peer interactions. A service simulator would provide the "lower layer" for peer interpreters, which are driven through their transitions based on canned event sequences. Specialized analysis tools could be defined which search the results to detect deadlock and partial deadlock occurrences. Final human analysis would be necessary to identify those areas which could cause real problems.

5.2.2 Service Conformance Validation

Another analysis task which focuses on peer interactions involves conformance between a mechanism specification and its corresponding service specification.

Such conformance validation at present requires the running an exhaustive set of service tests on the mechanism specification. Consequently, there is little difference between conformance validation during the analysis phase and the service testing of an implementation. Both involve the generation and execution of a set of service test sequences.

Generation of these tests can be through a combination of:

1. manual analysis of the service specification, generating "test templates" describing classes of tests

2. invocation of an automated test generator which generates a battery of related tests based on a test template

The automated test generator could choose test sequences which include checks on the "boundary conditions" of the template.

The "peer-interpreter/lower layer service simulator" configuration described earlier could then be used for conformance validation of the mechanism specification itself. Similar service tests could be executed upon implementations within an implementation testbed.

6. IMPLEMENTATION PERFORMANCE AND SERVICE TESTING

The laboratory contains an implementation testbed allowing the designers to examine performance characteristics of the proposed protocol mechanisms under a variety of network configurations.

6.1 DESCRIPTION OF THE NETWORK EMULATION

A major function of the laboratory will be the evaluation of protocols as implemented and executed upon the laboratory hardware. It is important that the evaluation be made of the protocol design and not of the specific implementation or of the environment in which it runs. To provide accessible and reliable performance measurements, the protocol under test should execute in isolation from all other software components of the test system. A monolithic system, one in which all software components (such as network emulators) execute in a single environment, does not provide the isolation desired for protocol evaluation. A distributed system, one in which multiple processor environments distribute the processing tasks, does provide an environment in which software isolation can be achieved (i.e. the protocol under test may reside in a dedicated processor). Additionally, a distributed system encourages the use of 'real' network hardware; the use of which provides realistic experience with protocols communicating over a network.

Hence, the backbone of the laboratory's implementation testbed will be a specialized local network capable of emulating a variety of network characteristics. Individual tests can be run under selectable network delay and reliability parameters.

To achieve such flexibility, the following requirements must be met:

1. The network must provide low delay, highly reliable, and high throughput channels - it is easy to increase a channel's apparent delay, decrease its apparent reliability, and decrease its capacity. Enhancements to a poor channel are more difficult to achieve.

2. The network must provide physically isolated individual channels to allow simultaneous tests within the laboratory. Here "physical isolation" means the following: tests running on one channel should not interfere with the delay and reliability characteristics of a second channel.

3. The network must include specialized devices which mediate target machine access to the channels. These devices will, under programmable control, alter the perceived channel characteristics in a manner appropriate for the desired test run. The devices should have access to all network channels, though not necessarily simultaneous access. In fact, it will be desirable to be able to isolate these devices from activity on a particular channel.

These requirements can be met by a wide-bandwidth broadband coaxial cable local network.

A broadband cable system for the protocol testbed can dedicate multiple frequency-derived channels for testing protocol implementations. A set of network adapters can be configured on the cable with the following characteristics:

1. Frequency-agile modems allowing programmed attachment and detachment from individual channels.

2. Adapter software implementing the "network emulation" functions such as increasing delays and unreliability of the channels.

3. Adapter software implementing a variety of link level (and possibly higher) protocols.

The 'network emulation' software, used to simulate a wide range of quality-of-service over the local network, will interface to the protocol implementation under test in a manner similar to that of a lower layer protocol. For example, this will allow the emulation of long haul packet switch networks or satellite links, during the testing effort. Functionally the test environment can be pictured as in Figure 2.

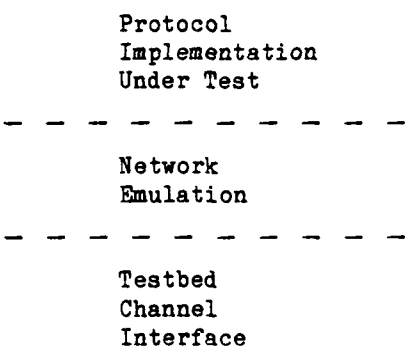


Figure 2: Test Environment

Within the test environment (pictured above) a separate development channel is also necessary. This channel is used to load the test unit with the software of the test version, to allow monitoring of ongoing tests and to provide control access to the protocol and emulation software during tests. Interfaces to this channel must be provided to both the protocol implementation under test and the network emulation software. Hence, the enhanced functional picture of the test environment is as shown in Figure 3.

Since the laboratory as a whole will require a local network, the broadband cable system used in the testbed could be shared with the other laboratory elements. In such a configuration, some channels would be used for general laboratory communication and some would be dedicated for implementation testing. By connecting both the testbed target machines and their adapters to the

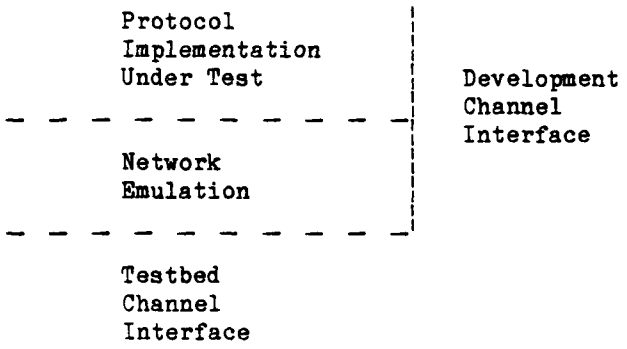


Figure 3: Enhanced Test Environment

general laboratory channels, these devices could be downline loaded and remotely monitored from other laboratory nodes.

The testbed target machine and its adapter will, hereafter, be referred to as a 'test unit'. Each 'test unit' will consist of a cable interface unit able to communicate over multiple channels and supporting two operating environments. The cable interface unit will allow separate access to both a development channel and a testbed channel. The multiple operating environments allow for greater isolation of the protocol under test (i.e. the target machine) and of the 'network emulation' (i.e. the adapter). Such a test unit is shown in Figure 4.

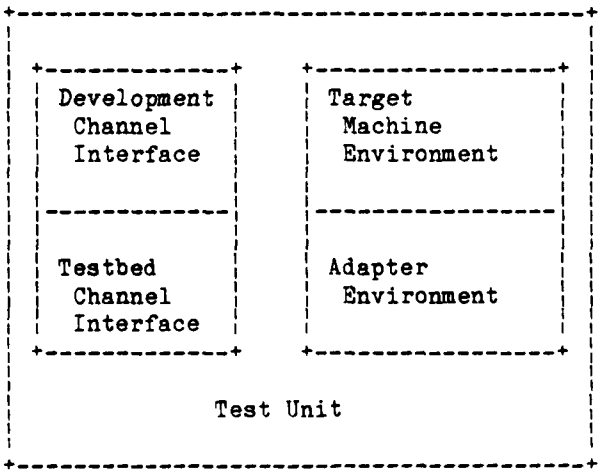


Figure 4: Test Unit

6.2 TEST UNIT HARDWARE REQUIREMENTS

The hardware implementation must satisfy the functional requirements of the 'test unit' while imposing the fewest restrictions upon protocol and testing operations. The functional requirements to be met are:

- o Communication Interface Unit, consisting of controller and modem, to allow connection to a local cable network on test and development channels.
- o Processor Environment to support the operating environments for the target machine and adapter.
- o [Optional] Mass Storage such as a small winchester disk for local storage of downline load images and test results.

Contentions for hardware resources, not associated with protocol operation, should be minimized even at the cost of duplicating hardware.

The 'test unit' needs to communicate with two separate frequency-derived cable channels:

- o the testbed channel provides the communication path between multiple protocol implementations undergoing test - this channel is required continuously during testing.
- o the development channel will be needed before testing to port the target and adapter environments to the test units; during testing to control the adapter software and to monitor protocol operations; and after testing to obtain results from both operating environments (e.g. protocol throughput statistics, memory dumps, etc).

The need for multiple, functionally separate, communication paths during testing argues for separate, independent communication interface units within the 'test unit'.

The 'test unit' will support two operating environments (software); one for the protocol under test (the target) and the other for the adapter software. For the purpose of protocol testing and evaluation, it is desirable to reduce the interactions between these different environments to a minimum. The minimum of interactions between these environments is the set of required interactions of one protocol, under test, with the next lower layer protocol. This minimum set of interactions is approached when the protocol under test is located in a separate hardware environment from the next lower layer protocol. Hence, it is desirable to implement the two operating environments upon two separate processor units.

While separate processor units are desirable for each operating environment, the requirements of each are similar. Each environment needs to supply services to the software implementations running within them (interrupts, timers, etc). In the future, higher level protocols will be developed and tested within the laboratory. These protocols will reside within the target

processors during testing and lower layer protocols, developed earlier in the lab, will reside in the adapter processors. The porting of these lower layer protocols from the target processor units to the adapter processor units is facilitated by having the different processor units duplicates of one another. Additional benefits accrue with the selection of duplicate hardware for the two processor units:

- o faster first implementation, support code for hardware and the operating system kernel is simplified (only single versions needed).
- o faster future implementations, porting of software from target to adapter processors is simplified (guaranteed) - hardware and compiler differences are eliminated.

[Optional] Local mass storage facilities may be desired, to be used for local sourcing of downline load of target and adapter processor images and to store large volumes of test results (multiple dump images, state transition tables saved upon each change, etc.). A small winchester type disk device should be adequate for this need. It should be noted that the need here is for large, reasonably fast access, data storage; no need is seen for and no provisions are to be taken to support a file system upon this device. The device chosen will be used for linear storage of data on a temporary basis. An alternative device for this purpose would be a small tape unit; the choice of a winchester disk has the advantage of high transfer rate and that areas of the disk may be accessed (skipped to) in a random (non-linear) manner.

A more detailed functional diagram of the test unit, showing communication paths between sub-units, is illustrated in Figure 5.

6.3 TEST UNIT SOFTWARE REQUIREMENTS

The requirements for the test unit software include providing an operating system kernel (for both the target and adapter environments) and adapter software in support of 'network emulation'.

For supporting functional isolation and transportability the two operating environments will be implemented on separate, duplicate hardware processor units. The software required by the target environment includes:

- o Operating System Kernel: interrupt service, timer service, memory management, scheduling, interprocess communication, etc.
- o Protocol under test (likely to be multiple processes).
- o Control/Statistical/Monitoring routines for control, test and evaluation.
- o [Optional] Controller and access routines for mass storage device.

The software required by the adapter environment includes:

- o Operating System Kernel: interrupt service, timer service, memory management, scheduling, interprocess communication, etc.

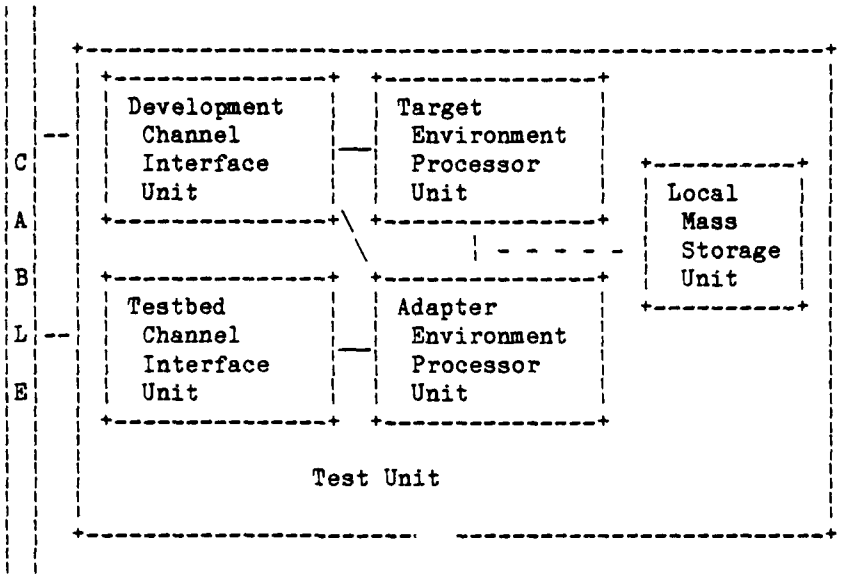


Figure 5: Test Unit Detail

- o Lower layer protocol (not under test).
- o Adapter routines to provide 'network emulation'.
- o Control/Statistical/Monitoring routines for control, test and evaluation.
- o [Optional] Controller and access routines for mass storage device.

6.3.1 The Operating System Kernel

The requirements of the two operating environments are similar. In the future, the protocols developed in the target environment will be required to operate within the adapter environment. This transportability requirement implies that the adapter operating system kernel be a super set, possibly minimal, of that of the target. Unless significant performance degradation accompanies the extra kernel services associated with the adapter, the same kernel will be used for both. Protocol transportability and ease of laboratory implementation will also be enhanced by use of a common operating system kernel.

The Operating System Kernel will support the following features:

- o Priority Process Scheduling with Preemption.
- o Message Queues.

- o Memory Management (allocation, deallocation and resizing).
- o Timer Services.
- o Interrupt Services.
- o Interprocessor Communication.

The operating system design chosen for the test unit processors is structured upon interprocess communication via message passing. Each process running in the system has a FIFO (first in first out) queue of messages associated with it. Each message in the queue represents either a request for service of the process or informs the process of an event of possible interest to it.

Memory management services support the message queues and buffers required to transport data across the protocol layers. Protocol data buffers place special requirements upon any system; a data buffer that descends the layers of a protocol is required to grow (headers associated with each protocol layer are appended with the original information); the reverse occurs when data is received by the lowest protocol layer and then ascends through multiple layers to a user. Three methods which handle protocol data buffer requirements are, as follows:

- o at each layer that appends or removes a header the entire message can be copied into a new buffer of the desired size.
- o as a message is created or received is copied into a maximum sized buffer, with room reserved for all possible protocol headers.
- o a message can be constructed of many pieces of data, at each protocol layer the required header data is appended to or removed from the complete message.

The first method is common, its advantage lies in that all sizes are known, buffers can easily be allocated from the memory pool; its disadvantage is in the overhead needed to copy the message several times during its passage through a protocol implementation. The second method is also common, it solves the problems associated with copying of the data, but it adds a disadvantage of fixed size buffers (many headers have optional fields; providing room for all options increases the overhead for all messages). The last method, the recommended choice for use in the test system, provides the advantages of the first two methods without their disadvantages; its own disadvantage, associated with its internal structure (a linked list of data segments), requires routines that access a message must be aware of this structure - adding to their complexity and overhead.

Interprocessor communication will be functionally similar to interprocess communication; a message passing system will be implemented using shared memory with handshaking or communication over an I/O channel, to processes the interface will appear as ordinary interprocess

6.3.2 Network Emulation Software Requirements

A necessary function of the operating system will be the support of "network emulation" and performance measurement facilities. One method for providing these facilities is through the addition of daemon processes within the task structure of the target environments.

Providing that a process structure based upon message passing is used in the implementation of the protocol under test (or in the network emulation software), the addition of a daemon process into the environment is quite simple: a daemon process is given the address of a process 'A' within the current environment (all messages normally addressed to 'A' will then be received by the daemon) the daemon preprocesses the queued messages as it so desires then forwards them to the real process 'A' where they are processed normally (e.g. a daemon process might simulate a less reliable network by rejecting queued messages containing notification of data received by a communications interface or a lower layer protocol).

The requirements of the network emulation software are dependent upon the specific protocol implementation under test. Development of network emulation software will proceed along with the protocol implementation it is intended to test and support.

6.3.3 Test Control Software Requirements

Test control software for the laboratory is needed for pre-test, during test and post-test functions. Pre-test requirements include compiled in instrumentation within the processor environments ported to the test units and routines to enable porting of the processor environments to the test units (or to the local mass storage units of the test units). During test requirements include the interactive control of the test environments, gathering of test data (with or without local storage) and test monitoring. Post-test requirements include the upline transfer of locally stored (in memory or mass storage) test results and of current processor memory images (for diagnosis or later reloading).

The routines for porting of the test unit environments are standard features of many systems; their development for this laboratory should not pose any problems. Specialized routines will be needed to support local mass data storage, local environment initialization from the mass storage device and upline transfer of test unit memory images and mass data storage of test results. An instrumenting compiler for the implementation language will be needed to help provide the performance information to be used in protocol evaluation.

Many of the desired features for test control and support will be determined as a result of actual experience using the laboratory; those listed here represent an attempt to foresee and provide needed functions prior to the establishment of the laboratory.

7. DEVELOPMENT OF THE FIRST IMPLEMENTATION

The tools described so far are intended to validate the consistency of the protocol's fundamental internal mechanisms. However, full confidence in a protocol design requires experience with an actual implementation. Here we describe the laboratory's support for the development of a "first implementation".

7.1 GOALS OF THE IMPLEMENTATION EFFORT

There are three basic goals driving the development of an implementation within the laboratory:

1. To permit testing of those aspects of the protocol's design which are intractable under symbolic execution.

2. To gain experience in the parametrization possibilities of the protocol's mechanisms under a variety of network characteristics.

3. To create a basis for the development of a "certifier" which can be used for acceptance testing of external implementations.

The previously described symbolic execution and automated walk-through efforts use a general set of programs which are capable of emulating any properly specified protocol, using the protocol's mechanism specification as input. Buffering, scheduling, and I/O details are ignored so that the abstract logic of the mechanism specification can be examined in a clear light.

The broad utility of these emulation tools is achieved at the expense of performance. This prevents a comprehensive examination of the timing-relevant aspects of the protocol's design. Timing problems (e.g. tempo-blocking) are analytically intractable and will typically be found only through an execution of the protocol over extended periods, under a wide variety of offered traffic. This requires implementations which are not so grossly inefficient that extensive testing in a "realistic" system environment is precluded. With multiple implementations, it is possible to observe the behavior of the protocol in its natural peer-to-peer relationship. Of course, one can simulate multiple machines on a single machine, but the veracity of such simulation can always be called into question.

A real implementation within the laboratory is also of great utility in testing the implementability of the specification document. By regulating the allowed contacts between a protocol's design team and implementation team, hidden assumptions and ambiguities within the specification will surface. Thus the first implementation effort is simultaneously a test of the protocol's mechanisms and of the specification document. Both aspects of this test should proceed in as "natural" an environment as possible.

Within the laboratory, a set of "target machines" will be used for testing the implementation. If realistic testing of a particular protocol requires special machines, such devices can be temporarily incorporated into the laboratory testbed. Although two target machines will suffice for two-party

protocols, certain multiple party protocols (e.g. gateway protocols, conferencing, or key distribution) will require three or more target machines. The use of a "delay and reliability tunable" local network between these processors will make possible the simulation of a wide range of network characteristics. The information gained through such testing can be used to determine parameterizations of the protocol's mechanisms for various real network environments. Gross inefficiences inherent in the protocol's design can be detected and corrected before the protocol specifications are released for external implementation.

As noted above the utility of emulation tools is obtained at the expense of overall system performance. The same is true of testing in general; the act of gathering test statistics for monitoring the protocol behavior may have an effect upon its performance and possibly upon it ability to perform. These effects must be considered in the evaluation of a test sequence and of the protocol implementation.

7.2 TOOLS

The development environment for the laboratory's protocol implementations could be structured in one of the following three ways:

1. All development could take place on the target machine.

2. Most development could take place on specialized development machines, with a final port to the target machines for testing.

3. Most development could take place on the same machine which is used for specification development and analysis, with a final port to the target machines.

We argue for the third alternative.

The first alternative has the obvious disadvantages of requiring substantial user support (editors, compilers) on the target machines. If specialized processors are to be used as target machines for a particular protocol, it is unreasonable to require that a full development environment be simultaneously provided. In addition, program development work would interfere with the primary testing mission of the target machines, since one protocol will be in the testing process while another is being implemented.

The third alternative has several advantages over the second. The existence of on-line protocol documentation could be used to advantage by the implementation team. Specialized conformance testing tools can be used both in the specification analysis and implementation testing efforts. And finally, the first implementation will in fact be part of the iterative protocol development as described in Section 2. The implementation language and development environment must be suitable for this type of development method, and there must be adequate source control facilities as previously described. Such code configuration control can be more easily integrated with the specification configuration control on a shared machine. For these reasons it appears to be most appropriate to include a code development environment for the

laboratory's target machines on the same processor as the protocol development and specification analysis environment.

7.2.1 Language

Although it would be best if the laboratory could implement protocols in a variety of languages for testing, such is not feasible. Even if such were possible, we would still be testing only a few data points in the large sample space of possible implementations. Consequently it is necessary that the laboratory standardize on a single implementation language - the natural choice is ADA. However, it must be recognized that this restriction in no way is intended to restrict the languages used in field implementations. [Note: Until such time as ADA becomes a widely available language it is possible that another structured high level language (C, PASCAL, etc.) will be used for initial development.]

7.2.2 Compilers

The development machine will require cross-compilers for each target machine within the laboratory. Such multiple compilers could share all but the final code generation software which must be specific to the target machine.

One of the code generators should produce code native to the development machine. This will allow some initial debugging to be performed on the development machine.

7.2.3 Debugging Tools

Two classes of debugging are required within the protocol laboratory. The first class of debuggers relates to a functional requirement for the establishment of the laboratory, that is the need to debug protocol designs at a specification (i.e. functional) level; this process, described in an earlier section, is accomplished through specification analysis and specification walk-throughs. The second class of debuggers, described in this section, deals with the debug of the actual protocol implementations to be exercised upon the target systems and used for performance testing and protocol evaluation.

Three types of debugging tools will be provided in the laboratory, for the protocol target implementations produced there, to be used with the target systems. It is possible that one or more of these tools will be available on the development system; the obvious choices are the basic debugger (supplied with the development systems underlying operating system) and a symbolic debugger supplied with a compiler generating code executable on the development system. The last debugging tool (Control and Monitoring routines) is to be supplied as support services of the target operating system and would only be available on the development system within a target system emulation.

The most basic debugging tool is supplied with the hardware upon which the code will be run (development or target hardware). These traditional 'debuggers' allow step-by-step and controlled execution of the machine code image; though tedious and time consuming these 'debuggers' have been

successfully used for many years.

The next debugging tool is the 'symbolic debugger'; like the traditional debuggers they allow step-by-step (i.e. line-by-line) and controlled execution of the generated code. Symbolic debuggers differ from traditional debuggers in that debugging proceeds not on a machine instruction level but on a high level language line-by-line level. Debugging is faster and is accomplished on a functional level, independent of the machine in which the routine may eventually execute.

The last debugging tool is supplied as routines incorporated within the software of the protocol implementation. Control and Monitoring routines may be incorporated within the protocol implementation image ported into the test unit processors. The protocol implementer will then be able to control the operation of the test unit processor from the development system.

7.2.4 Instrumentation

For supporting the testing effort multiple instrumentation facilities will be available to the protocol implementer. These facilities will be supplied by the compilers, the operating system kernel and user routines.

The compilers will offer user-selectable "profiling" options. Code generated under such options will include special debugging and statistics gathering routines. This will allow uniform testing of buffer utilization, processing delays, and other performance aspects of the protocol implementation(s).

The operating system kernel will support selectable information gathering routines for reporting on the performance of the protocol implementation(s) within the operating system environment. Statistics on memory usage, scheduling frequencies, delays, etc. will be supplied upon request.

The operating system will be designed in such a manner that user written 'daemon' processes can be easily introduced within the task structure. The protocol designer will be able to write specific monitoring routines to fit within his protocols task structure to gather specialized information on its performance.

8. CERTIFYING EXTERNAL IMPLEMENTATIONS

A primary goal of the protocol laboratory is to provide mechanisms for certifying external implementations of a standardized protocol. This section reviews the purpose of certification, outlines possible approaches, and presents some requirements for laboratory tools supporting this activity.

8.1 PURPOSE OF CERTIFICATION

The ultimate aim of "external certification" is to ensure that two external implementations of a standardized protocol can communicate with each other via practical lower level protocols and communications facilities. For example, one would like to guarantee that an FTP protocol on one machine can communicate correctly with another machine's FTP protocol given that the lower level protocols and communications media between the two provide the required services.

This certification goal puts constraints on the protocol service, lower layer, and mechanism specifications. First, the mechanism specification must be "complete" so that the protocol's correct functioning is completely independent of the implementation of discretionary items (timeout values, interface flow controls, for example). Also, the service specification must contain some "progress" specification and describe performance in terms of the lower layer provided service.

8.2 CERTIFICATION METHODS

The certification procedures applicable and the degree of "coverage" provided will be highly dependent on the testing facilities available at a remote site. A practical approach is to standardize on a small number of test procedures for different specified test facilities varying from primitive to sophisticated.

8.2.1 Basic Facilities

The basic method tests the protocol "in-situ". It assumes the existence of "proven" lower level protocols at both the external and laboratory site such that the laboratory "standard" can be connected to the external protocol through them and some transmission medium. A test driver is necessary to control the protocol-under-test's user interface. This driver connects to a test generator program in the protocol laboratory. This connection may be "out-of-band" (e.g. a dial-up line). The test generator also connects to the laboratory standard's service interface and can run service tests to ensure the two protocols can provide the services specified.

This approach is useful to determine raw compatibility between two implementations, and given that one is a "standard" gives some degree of confidence about the correctness of the external implementation. However, it does not provide very complete testing, and certainly does not verify that the external protocol will work with other external protocols, or that it would work with other lower layers, or other media. Also, the availability of "proven" lower layers for the test set-up may restrict testable external implementations.

8.2.2 Advanced Facilities

More advanced tools can be used at the external site to enhance the conformance validation process. In particular, one thing missing with pure service testing of the external implementation is knowledge of the lower level system's actual behavior during the test run. For example, one should distinguish between problems which arise due to mistakes within the external implementation itself and those which are caused by spurious malfunctions within the network.

For this purpose it is necessary to monitor the activity at the lower level interface of the protocol implementation under test. The results of such passive monitoring can be communicated to the laboratory after the test run and used during the analysis of the complete test results. This allows a deeper examination into the behavior of the implementation under test.

For these test runs, it is necessary not only to provide a driver at the upper interface of the implementation under test, but also a passive monitor at the lower interface. Both the driver and the monitor could be controlled from the laboratory via an out-of-band channel.

8.2.3 Sophisticated Facilities

A more sophisticated approach employs more elaborate test facilities at the external site. Two test drivers must be available on the remote host: one connected to the user interface, and the other at the low level interface. These drivers are state machines that generate output based on both a stored test sequence and received interface events. Both the test sequence and state machine description are down-loadable from the protocol laboratory, and will record the responses from the protocol under test for later transmission to the laboratory and subsequent analysis. The test stimuli would in general be conditional, based on clock time, implementation responses, or both. The results would include timestamped stimuli and responses. The tests generated by these drivers will be very similar to those used to validate the initial laboratory implementation.

The coverage, that is the probability of catching a malfunction, using this method will be quite high, and the data gathered can be used in future certifications. This data should enable future certifications using the sophisticated method to simulate the interconnection of two external implementations. That is, the response from one protocol will be used to stimulate the other and vice versa.

Both the generation of stimuli and analysis of results should be automatically generated from the mechanism specification. Certain tests may also require the results from previous tests as timeouts cannot be controlled, but must be deduced from certain test results. The idea would be to run the implementation through all possible internal states, and measure internal parameters such as timeout values and storage limitations. Once such a test suite was run and recorded, the results could be used to simulate the external implementation. Then at a later time, two such implementations could be simulated, interconnected, and their service level parameters (throughput, etc) measured

in the laboratory.

8.3 LABORATORY TOOLS FOR IMPLEMENTATION CERTIFICATION

The tools required to support the certification effort must provide for the development of a "standard" implementation, test scripts, test analysis, and test administration.

8.3.1 External Implementation Tester

The only approach feasible for large-scale certification of external implementations involves "service testing" of the implementation against a standard implementation. In service testing, the only interface available to the test software is the protocol implementation's "user" interface - test software has no control or monitoring capability over the system or lower level interface.

Unfortunately, there is no guarantee that service testing will exercise all of the protocol's internal mechanisms. For example, it may important to verify that the implementation behaves correctly in response to a received message containing an item which is "randomly" generated by its peer. Such would be the case in TCP, which incorporates a randomly generated initial sequence number into the connection request message. It would be desireable to validate that the external implementation responds correctly not only to a "random" value of 4662, but also to the specific value of 0.

This places the following requirement on the standard implementation: various "internal" actions (such as generation of a random initial sequence number) must be controllable by the test driver. Consequently it is not sufficient to take a "normal" implementation (assumed to operate correctly) and use it as the certification standard, since test drivers which can only act as a user of the protocol will not have the degree of control that they require. The standard implementation used for certification thus must not only be a correct implementation, but it must also be a specialized implementation.

The protocol implementation used as the standard can be derived from the implementation used within the laboratory's internal testbed. Such a derivation may involve additional parameterizations which together can give a test driver the desired level of control. Some monitoring code may be removed, and other monitoring code may be added.

Thus the testing of external implementations will involve external test drivers which are ordinary users of the protocol, but the test drivers at the standard implementation exercise a more thorough level of control over its operations.

8.3.2 Test Script Generation

The semi-automated generation of service test sequences for conformance validation was discussed in the earlier section on "Protocol Analysis". These test sequences can form the basis of the certification test procedures.

The standard implementation provides a richer interface to its test drivers than does the external implementation, and this will be advantageously used during certification. The manner in which this is done is protocol-dependant, and no general laboratory tools supporting this can at present be identified.

8.3.3 Certification Test Administration

Certification testing requires external access to the standard implementation. Since the laboratory is not to be "open" to the external world, the laboratory should provide specialized external access to a "certification device" which is tailored for this function. Such access can include a variety of networks and special interfaces.

Certification is a formal procedure, and requires a scheduling policy. This activity is completely independent of the laboratory's protocol development activities, and could in fact be distinct from the laboratory. However, incorporation into the laboratory is desired since the development and initial validation of the standard would benefit from the laboratory's tools.

The problem of "global configuration management" will arise when a set of external implementations are certified. The laboratory could provide technical support for this task, using the centralized certification facility to enforce configuration policy decisions.

9. LABORATORY MANAGEMENT

The Protocol Laboratory will be a substantial computer installation, supporting simultaneous development of multiple protocols. Its role as a certification facility requires that it be remotely accessible; yet it must not be completely "open" to the external world. These considerations require a comprehensive laboratory management system.

9.1 CONFIGURATION MANAGEMENT

As a protocol development effort progresses within the laboratory, a wide variety of specifications, protocol software, test plans, test results, and other documentation must be maintained. This calls for a configuration management policy and tools to support its enforcement. In addition, the laboratory itself will include an evolving set of tools. These must be maintained under a configuration management system in a manner independent of any one protocol's development.

Independent of the approach used to design protocols, tools and conventions for maintaining a correspondence between versions of the service and mechanism specifications, the implementation, the test results, and all related miscellaneous files must be provided. These conventions must be formalized to such an extent that they cannot be overridden. For example, it must not be possible to do a quick "off the record" test involving a change to any specification, or implementation. The tracking system must allow full recovery to any previous state, and provide a quick reference system to past work including dates and authors / experimenters.

A basic foundation for the establishment of the protocol laboratory's configuration management system can be obtained from the DoD 'Stoneman' report for the 'Ada Programming Support Environment (APSE)' [5]. Although the initial laboratory will certainly not include a standard APSE, it may be desirable for it to evolve towards such a system. Such evolution could be encouraged if the configuration management tools provided within the protocol laboratory are compatible with the APSE requirements. In any event, the APSE configuration management requirements seem to address most of the laboratory's concerns. The specifications within the Stoneman document should be viewed as a guide, not a constraint, in the development of the lab support environment.

Specific attention should be given to implement a database system functionally similar to that specified in the report (APSE Database). The first step in creation of this desired database concept is a source document configuration and control system, similar to PWB/UNIX Source Code Control System (SCCS) [6]. The abilities of SCCS need to be expanded to allow for the coordinated control of sets of related source documents (i.e. documentation, specifications, code source files, target image, test sets, test results, etc.). In an APSE Database, relations are maintained among the various objects (code, specifications, test results, etc.). Such relationships are not explicit in an SCCS-type system, which only maintains history information on individual objects and controls object access.

9.2 SUPPORT FOR GLOBAL CONFIGURATION CONTROL

Once in full operation, the DoD Protocol Standardization Program will lead to a proliferation of protocol implementations scattered throughout the DoD internetwork. Management of these implementations involves "global" configuration control, as updated versions of protocol specifications will be periodically released and implemented. Such configuration control is the responsibility of the Protocol Configuration Control Board (PCCB).

The protocol laboratory can serve as the focal point for this global configuration control. In this way, the laboratory's roles of protocol development and protocol certification are extended to cover the entire life cycle of individual protocols. Although such a role is independent of the laboratory's primary development and certification activities, it may reasonable to include a global configuration control database within the laboratory.

9.3 RESOURCE SCHEDULING AND ACCESS CONTROL

The laboratory must be externally accessible for certification and perhaps to allow remote use of specialized software tools. Also, initial implementation testing may also be desired using external networks in addition to the laboratory internal testbed. Connections with ARPAnet, EISN [7], EDN [8] and other DoD networks are anticipated. However, the laboratory is not to be an "open" system. Unrestricted remote access to laboratory components could compromise the laboratory's protocol testing efforts.

Laboratory resources will be shared by a variety of concurrent projects. However, each project (e.g. a protocol testing effort) requires the exclusive use of certain laboratory resources such as target machines and testbed channels. Consequently, the laboratory requires a scheduling policy and appropriate protection mechanisms on the resources for enforcement.

Detailed requirements for scheduling tools cannot be defined until policy issues covering remote access and resource reservation are resolved. It seems that a common mechanism covering both remote and laboratory-internal use of resources is desired.

10. SUMMARY OF LABORATORY REQUIREMENTS

The DoD Protocol Laboratory will provide support for the design of new protocols, development and validation of protocol specifications, testing of initial protocol implementations, and certification of external protocol implementations. The desired goals can be achieved if the laboratory meets the requirements outlined in this report, which are summarized below:

1. It must support and even encourage the creation of early implementations of a protocol within a controlled development environment, providing performance and implementability feedback to the protocol designers. Iterations between service specification, mechanism specification, and implementation will be managed under a comprehensive laboratory configuration control system.

2. An interactive environment providing automated support for protocol specification walkthroughs will ease the mechanism validation process. Both single-interpreter and peer-interpreter configurations can be invoked. A general state-machine interpreter can form the basis of this environment, based on the DoD formal mechanism specification technique. Peer-interpreter walkthroughs use a service simulation of lower layer protocols, derived from the the DoD formal service/interface specification technique.

3. Analysis of a protocol's consistency, state-reachability, and freedom from deadlocks can be supported through a set of computation-intensive tools. Exhaustive search techniques can use the general state-machine interpreter which is also used for walkthroughs. Additional "symbolic processing" tools will provide reachability analysis support.

4. An implementation testbed will be based on a high-bandwidth local network. Specialized network adapters will selectively degrade the local network's characteristics to emulate real DoD networking situations. Multiple local network channels are required for testing of multipeer protocols (e.g. gateway protocols or key distribution protocols). These channels must be independent for test isolation. An additional local networking capability is required for general support of laboratory protocol development activities such as downline loading, file transfers, and test monitoring. A broadband cable network can provide the desired multichannel capability for both the testing channels and the development channels.

5. A software development system for protocol implementations must be provided. This must include cross-compilers for the laboratory testbed target machines. Source configuration control must be integrated into the laboratory configuration management system which covers specifications and other supporting documents.

6. A "standard" implementation must be derivable from the laboratory protocol implementations for use as a certification tool. Remote access to the standard must be provided so that external implementations can be tested for conformance.

Finally, the Protocol Laboratory must be capable of evolution. At each stage in its evolution, it must provide a coherent set of related tools which together make a substantial contribution to the protocol development process. The initial operational capability may include only those tools supporting initial implementation testing, such as the testbed and its development system. Additional enhancements could then be added to support the protocol design, validation, and certification efforts. It is clear that at no point in its evolution would the laboratory be considered "complete".

11. REFERENCES

1. DCEC Protocols Standardization Program, Preliminary Architecture Report, Sytek Staff, System Development Corporation TM-7038/200/00, 1981.

2. G. Simon, DCEC Protocols Standardization Program, Draft Protocol Specification Report, System Development Corporation, TM-7038/204/00, 1981.

3. W. Teitelman, Interlisp Reference Manual, Xerox Palo Alto Research Center, October 1978.

4. T. Teitelbaum and T. Reps, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, CACM 24/9, September 1981

5. Requirements for Ada Programming Support Environments, "Stoneman", Department of Defense (USDR&E), February 1980

6. M. Rochkind, The Source Code Control System, IEEE Trans. Soft. Eng., SE-1 No. 4, December 1975

7. Experiment Plan for the Wideband Integrated Network, MIT Lincoln Laboratory, December 1979

8. EDN System Approach Report, Report No. 3770, Bolt Beranek Newman, April 1978.


DATE
FILME
8